Capítulo 6

Programación funcional

6.1. Introducción

Al desarrollar software se identifican paradigmas de programación que gobiernan el enfoque con el que se encara el problema, se diseña una solución y finalmente se redacta el código fuente. Entre los paradigmas más utilizados se destacan el paradigma estructurado y el paradigma orientado a objetos.

Otro paradigma de uso habitual es el paradigma funcional. El mismo tiene una base matemática bastante compleja y por tal motivo existen pocos lenguajes denominados puros, es decir, que permiten únicamente desarrollar con este paradigma. Sin embargo, actualmente todos los lenguajes de uso cotidiano (tales como Java, C#, Python o Javascript) incluyen ciertos elementos de esta metodología.

A pesar de que estos lenguajes están mayormente concentrados en los paradigmas estructurado y orientado a objetos, ofrecen algunas características de programación funcional. Esto se logra mediante la posibilidad de trabajar con un nuevo tipo de dato denominado "función".

El origen de esta perspectiva puede parecer desconcertante, pero se la puede analizar desde el punto de vista de las variables. Cuando se utiliza la programación estructurada, los programas poseen variables donde se almacenan valores. En esta modalidad, cada variable tiene la capacidad de almacenar un único valor de alguno de los tipos de datos ofrecidos por el lenguaje, tales como números, cadenas, booleanos, etc. En cambio, en la programación orientada a objetos, las variables pueden contener objetos, los cuales pueden tener múltiples atributos y métodos. Aunque es válido argumentar que las variables solo contienen referencias a objetos, en esencia, sigue siendo cierto que cada variable almacena un objeto.

En la programación funcional, las variables almacenan código fuente, es decir, conjuntos de instrucciones del lenguaje. Por lo tanto, si una variable se declara como una función, es posible ejecutar el código que esta contiene. Y dado que son variables pueden cambiar su contenido, por lo tanto, si en algún momento se asigna un código diferente a la variable y se ejecuta su contenido nuevamente, se llevará a cabo una ejecución distinta. En conclusión, siempre se ejecutará el conjunto de instrucciones que esté almacenado en la variable en ese momento específico.

6.2. Funciones de orden superior

Los lenguajes que ofrecen programación funcional, incluso en su forma más básica, brindan la posibilidad de tener variables cuyo contenido sea una función. También se pueden tener parámetros de tipo función, lo cual implica que un método puede ejecutar una función sin necesidad de saber con exactitud su comportamiento.

De la misma manera, una función o un método de una clase puede retornar una función para que sea almacenada desde la llamada al mismo.

Los métodos que poseen parámetros o retorno de tipo función se denominan métodos o funciones de orden superior. Esta noción tiene una gran relevancia matemática y es muy útil en diversas situaciones de programación.

6.3. Variables de tipo función

Una variable puede almacenar una referencia a una función. En el siguiente ejemplo se declara una variable llamada `imprimir` y se asigna su valor con el nombre de la función `print`. Debe notarse que no se utilizan los paréntesis que efectivamente iniciarían una llamada a la función print. Luego de la asignación, la variable puede ser ejecutada y el resultado es que se ejecuta un print.

```
imprimir = print

imprimir("Hola, mundo!")
```

6.4. Colecciones de funciones

De la misma manera que una variable puede almacenar una referencia a una función, se pueden crear colecciones cuyos elementos sean funciones.

En el siguiente ejemplo se crea una calculadora básica con las cuatro operaciones elementales de la aritmética implementadas con una función por cada una. Para que el usuario pueda seleccionar la operación deseada se ofrece un menú con un número natural por cada opción. Al momento de ejecutar la función correspondiente se accede a una lista que contiene en cada posición una referencia a la función correspondiente en el menú. Esto permite acceder a la lista usando como índice la opción seleccionada, y dado que el valor retornado es efectivamente una función se la puede ejecutar pasándole los parámetros que la misma requira. Para ello la lista `operaciones` es inicializada con un valor `None` ya que no hay ningúna operación con número 0 en el menú, y luego cada una de las referencias a las funciones de las operaciones aritméticas.

```
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b

def multiplicacion(a, b):
    return a * b

def division(a, b):
    return a / b

def ejecutar_calculadora():
    operaciones = [None, suma, resta, multiplicacion, division]
    print("Calculadora básica")
    print("Seleccione una operación:")
    print("1. Suma")
    print("2. Resta")
    print("3. Multiplicación")
    print("4. División")
    print("0. Salir")

    opcion = int(input())

    while opcion ≠ 0:
        if opcion ≥ 1 and opcion ≤ 4:
            a = float(input("Ingrese el primer número: "))
            b = float(input("Ingrese el segundo número: "))

            resultado = operaciones[opcion](a, b)
            print("El resultado es:", resultado)
        else:
            print("Opción inválida. Intente nuevamente.")

        print()
        print("Seleccione una operación:")
        opcion = int(input())

ejecutar_calculadora()
```

6.5. Funciones lambda

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y concisas que se definen sin un nombre y se utilizan en el momento en que se necesitan. A diferencia de las funciones normales, que se definen utilizando la palabra clave `def`, las funciones lambda se crean utilizando la palabra clave `lambda`.

La sintaxis básica de una función lambda es la siguiente:

```
lambda argumentos: expresión
```

Donde lambda es la palabra clave que indica que se está creando una función lambda, la lista de argumentos son los parámetros de la función, separados por comas pero sin los paréntesis que exige el bloque def y la expresión es la que se evalúa y devuelve como resultado de la función.

Una función lambda está limitada a una sola expresión y no puede contener múltiples líneas de código. En el caso de necesitar realizar múltiples operaciones, ejecutar varias líneas de código o usar estructuras de control, debe desarrollarse una función regular con def.

En el ejemplo anterior de la calculadora se pueden evitar todas las definiciones de las funciones suma, resta, multiplicacion y división de la siguiente manera:

```
suma = lambda a, b: a + b
resta = lambda a, b: a - b
multiplicacion = lambda a, b: a * b
division = lambda a, b: a / b
operaciones = [None, suma, resta, multiplicacion, division]
```

6.6. Secuencias y operaciones funcionales

Existen funciones de orden superior que simplifican la manipulación de los datos de las secuencias.

6.6.1. Filter

La función `filter()` recorre una secuencia y ejecuta por cada uno de los datos de la misma una función recibida como parámetro y que se espera que devuelva un valor de verdad, tanto valores booleanos como convertibles a boolean (truthy). Esta función retorna una nueva secuencia cuyos valores son los mismos de la secuencia original pero únicamente aquellos para los que la función retorna un valor verdadero.

La sintaxis de filter es:

```
filter(funcion, secuencia)
```

Dado que filter retorna una nueva secuencia, la misma debe ser recorrida con una instrucción for o convertida a una colección, con las funciones `list()` o `set()`.

Por ejemplo, la siguiente llamada a filter extrae los números pares de una lista:

```
es_par = lambda x: x \% 2 == 0
numeros = [2,1,35,8,5,21,4,6,9,6,3,2]
pares = list(filter(es_par, numeros))

print(pares)
#[2, 8, 4, 6, 6, 2]
```

6.6.2. Map

La función `map()` permite generar una nueva secuencia de forma tal que cada elemento se obtenga con el resultado de ejecutar una función a cada elemento de la secuencia original. La sintaxis es idéntica a la de la función filter, de forma tal que el primer parámetro debe ser la función a repetir y el segundo contiene la secuencia que debe procesarse.

En el siguiente ejemplo se obtienen una lista con los cuadrados de los numeros de la lista original:

```
numeros = [2,1,35,8,5,21,4,6,9,6,3,2]
cuadrados = list(map(lambda x: x ** 2, numeros))

print(cuadrados)
#[4, 1, 1225, 64, 25, 441, 16, 36, 81, 36, 9, 4]
```

6.6.3. Reduce

La función `reduce()` recorre una secuencia aplicando una función de pliegue o reducción, obteniendo como resultado un único valor. Para la operacion de reducción requiere una función de dos parámetros que realice alguna operación con los mismos y retorne un valor. Al ejecutarse toma los dos primeros elementos de la secuencia e invoca a la función recibida, la cual debe retornar un resultado. A continuación invoca nuevamente a dicha función enviándole como parámetros el tercer elemento de la secuencia y el resultado de la primera ejecución. Luego continúa invocando la función con cada uno de los restantes elemntos de la secuencia y el retorno de la ejecución anterior.

Por ejemplo, si se dispone de una función suma, que reciba dos números y retorne la suma aritmética de esos parámetros, puede utilizarse reduce de la siguiente manera:

```
numeros = [23,6,9]
total = reduce(suma, numeros)

print(total)
# 38
```

Al ejecutarse reduce, inicia invocando a la función suma con los dos primeros valores de la lista (23 y 6), obteniendo como resultado el número 29. A continuación vuelve a llamar a suma pasandole como parámetros 29 (el resultado anterior) y 9 (el tercer elemento de la lista), obteniendo 38. Y dado que la lista no posee más elementos retorna ese valor como resultado final de la reducción.

La función reduce fue movida al módulo `functools` por lo tanto para poder utilizarla debe ser importada con `from functools import reduce`.